# Leveraging Task Contexts for Managing Developers' Coordination

Kelly Blincoe
Drexel University
Department of Computer Science
3141 Chestnut Street
Philadelphia, PA 19104
kelly.blincoe@drexel.edu

Giuseppe Valetto
Drexel University
Department of Computer Science
3141 Chestnut Street
Philadelphia, PA 19104
valetto@cs.drexel.edu

Sean P. Goggins
Drexel University
College of IS&T
3141 Chestnut St.
Philadelphia, PA 19104
sgoggins@drexel.edu

## ABSTRACT
We introduce a new method for determining work dependencies that are antecedents to *coordination requirements* among members of a software development organization. Our method leverages records of individual activity associated with development tasks, sometimes called *task context*, which can be collected by monitoring the actions carried out by a developer during work sessions within her development environment. We describe an algorithm that measures similarity between task contexts and produces a measure of *closeness* between the corresponding developers. By means of a field study on an open source project that routinely records task context data, we show how the closeness relationship accurately determines the same coordination requirements detected using traditional methods. Our method also provides a temporal advantage, since it uses "live" instead of historical data. We explain how these findings make coordination requirements actionable for management-, design- and team-related decisions as the development work is underway. This moves research in this area from post-mortem analysis to proactive detection of coordination requirements.

## Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *Programming Teams, Productivity.*

## General Terms
Management, Measurement, Human Factors.

## Keywords
Awareness, Closeness, Coordination Management, Coordination Requirements, Socio-Technical Factors, Task Context, Tools.

## 1. INTRODUCTION
The coordination of concurrent activities by multiple developers during a project remains problematic for software development organizations. Software engineering pioneers such as Parnas [29] and Brooks [3] recognized the importance of efficiently managing the work dependencies and coordination overhead arising within a development team. Contemporary software development trends toward larger and more widely distributed teams and projects exacerbate this problem [5][19][20][21].

Coordinating the work of software developers on complex projects continues to be a challenge in industry and the open source community because the task dependencies and Coordination Requirements (CRs) that emerge between developers are difficult to detect. Cataldo et al. [6] [8] introduced a framework to detect and quantify CRs between pairs of developers by first identifying the technical dependencies between software artifacts modified during their assigned tasks. This formalization of CRs enabled the introduction of the Socio-Technical Congruence (STC) index to measure the quality of coordination in a software project. This is analogous to what is done in the domain of product design [4][18][32]. High levels of socio-technical congruence seem beneficial. Empirical studies show that when the flow of coordination among developers is increasing in accord with detected CRs, productivity is likely to improve [8][32].

Unfortunately, current methods for detecting CRs and calculating socio-technical congruence suffer from two serious drawbacks. First, current methods do not allow developers to be aware of CRs during their development work. This is because CRs are identified by mining the source control repository of the project for changes to artifacts committed by a developer. Thus, that data is often amenable only for post-mortem analysis of CRs. Second, committing a change is the culmination of the knowledge-intensive process carried out by a developer to complete her task. The committed artifacts available by mining source repositories are, therefore, often only a subset of the complete *working set* of artifacts used by the developer to gain knowledge of and complete the task.

Without a "live" view of activities, CRs and STC are not yet actionable devices for managing coordination in software projects. Several potential applications for such a live view have been discussed. For example, Ehrlich et al. have elaborated a way to rank CRs in a project, enabling prioritization of those whose resolution can improve socio-technical congruence the most [15]. Valetto et al. have proposed a set of management-, design-, and team-related decisions that can be used as alternatives to resolve CRs; each with its associated costs and risks [37]. These and other techniques require the ability to detect, analyze and manipulate CRs as they emerge from the dynamic activities carried out concurrently by developers.

To surface CRs in a timely way we introduce a new method for detection, which leverages different information. Instead of considering the technical dependency relationship between the artifacts committed in different tasks, we consider the full working set accessed throughout the tasks and determine a *closeness* relationship, which predicts similarity between developers' working sets.

Our method requires a facility that monitors developers' actions on artifacts, as they occur, such as that made available by the very popular Mylyn framework (formerly Mylar) [23][24]. By means of a field study, we describe a method that not only detects CRs at least as precisely as the traditional method, but also detects them

substantially earlier. We show that detection is possible while the development work is still underway and often only shortly after the onset of impacted tasks. Thus, our method makes CRs actionable for project governance decisions and effectively supports the management of coordination in software projects.

The rest of the paper is organized as follows: in Section 2, we discuss background material and related work; in Section 3, we present our method, including the data we use and the algorithms we have conceived to determine closeness; in Section 4, we describe in detail the setting, method and results of our empirical study; in Section 5, we discuss the significance of our contributions, including its possible limits and likely applications; finally, in Section 6, we offer some concluding remarks.

## 2. RELATED WORK

In the course of a project, it is paramount to be able to identify and manage work dependencies. If such dependencies are not recognized, the costs of reconciling conflicting work often lowers product quality and developer productivity [3][7] [20][32].

The design literature, beginning with Parnas' recognition of the workflow implications of modularization [29], equates modules with work items and focuses on ways to streamline the structural interdependency of modules as a way to maximize task parallelism [1][32][35]. In the case of software, such structural dependencies are often derived from a syntactic analysis of the code base. The software evolution literature offers an alternative to identify technical dependencies. Gall et al introduced the notion of logical coupling, based on the "files changed together" heuristic [17], which aims at identifying semantic relationships that may not manifest in the programmatic implementation of the software. Whereas syntactic dependencies exist *a priori* with respect to a project and an organization, logical dependencies reflect accumulated empirical evidence about how the development work unfolds in the project.

This shift, from studying technical dependencies *per se* to exploring the complex interplay they have with organizational structure and project dynamics, continues in recent research on the socio-technical aspects of software engineering. For example, several field studies on development teams show how the barrier of information hiding may imperil the ability of team members to coordinate with one another and remain aware of important work decisions [5][12][19]. Herbsleb et al. [22] argued for a systematic framework to satisfy, as opposed to simply eliminate, work dependencies, by explicitly orchestrating coordination within a project. Works on Coordination Requirements and Socio-Technical Congruence in software development are part of this thread of research [6][7][8].

Cataldo et al. [8] describe STC as an index of the quality of coordination, based on the alignment between team interactions and technical dependencies in the project. Conway [9] was the first to describe the possibility of such an alignment. STC measures the extent to which coordination requirements and coordination behavior are aligned in practice. STC is expressed as a simple ratio between CRs that are satisfied by actual acts of coordination (typically communications) [25] and the set of outstanding CRs between developer pairs, according to the following formula:

$$CR = TA \times TD \times TA^{t}$$

In this formula, TA is a people-by-task matrix representing task assignments, and $TA^t$ is its transpose; TD is a task-by-task matrix capturing the work dependencies between tasks. Those, in turn, are established by considering the technical dependencies occurring between artifacts involved in those tasks. According to this formula, a CR between two developers Alice and Bob can be represented graphically as in Figure 1, where arc $TD_{ab}$ represents a technical dependency between software artifacts $S_a$ and $S_b$, which are involved in tasks to which Alice and Bob, respectively, are assigned (denoted by arcs $TA_a$, $TA_b$).

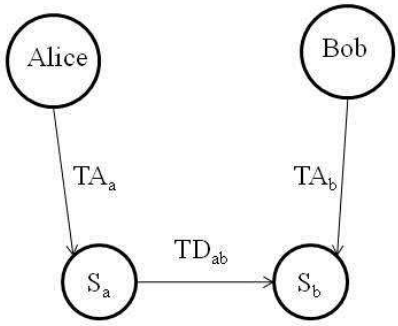


**Figure 1: Representation of a Coordination Requirement.**

There are several methods for computing technical dependencies. Cataldo et al. offer empirical evidence that logical coupling, obtained by tracking files checked in together, provides a more reliable representation of the technical dependencies relevant for CR detection than syntactic coupling does [6]. Logical dependencies, however, are computed based on past project history and become visible only after work is completed. Even syntactic dependencies used to compute CRs, as done by Ehrlich et al. [15], only become fully known after the fact. Similarly, the association of developers to artifacts typically becomes visible only after the tasks are completed and changes to those artifacts have been committed to the software repositories of the project.

All of the above impede the use of CRs as an actionable concept and limit their potential to help management coordinate work while it is happening. Our research aims to close this gap. In this respect, it contributes to research on CR visibility in software engineering and the more general construct of awareness in complex software development efforts.

Awareness [14] is one of the ways used to conceptualize coordination, and awareness research in software engineering has a rich history. de Souza and colleagues identified four major software engineering problems related to awareness: managerial awareness of social dependencies, developer awareness of code dependencies, locating needed expertise and locating similar expertise [11]. Many tools and approaches have been developed to address one or more of those problems. For our purposes, we can classify them based on whether they rely upon the analysis of historical data or on mechanisms for sharing of the developers' workspaces.

Many of the tools that rely on historical data focus on the problem of expertise location. Examples include Expertise Browser [28] and Hipikat [10]. Expertise Browser mines the source code repository to implement recommendations on experts across a globally-distributed software development team. Hipikat maintains an exhaustive "project memory" about documents, files, messages, people and change tasks. It then searches for keyword, temporality or social commonality across these dimensions. Given an artifact of interest, Hipikat generates a similarity score for

other elements kept in the project memory using a data-type-specific recommender.

Some of the awareness tools that rely on historical data address issues that are more directly related to Coordination Requirements. For example, Ariadne explicates social and code dependencies arising in a team through various visualizations of the team as a social network [11]. While the use of social networks has been widely explored also in other awareness approaches [26], Ariadne, stands out since the arcs in the network are computed in a way similar to the CRs computation introduced by Cataldo [8]. Ariadne uses syntactic couplings to represent artifact/artifact dependencies and mines developer/artifact associations from the commit records in the source code repository. Therefore, it suffers from the drawbacks related to reliance on post-mortem data we discussed above.

EEL combines the concepts of working set and CRs. It builds an ego-network for each developer based on the current artifacts in her working set and their logical couplings to artifacts committed in the past by other developers [27]. Although the goal is quite similar to ours, the main difference is that we want to bypass the technical dependency relationship altogether for CR determination. Since EEL relies on such dependencies, it captures past – as opposed as impending – CRs.

Tesseract tries to explicitly incorporate the concepts of CRs and STC for awareness purposes [31]. It presents users with four distinct panels, dealing with project activity, artifacts, developers, and tasks which are integrated through the notion of congruence. Although those connections remain rather implicit, Tesseract perhaps goes further than any previous tools in achieving this vision. Also for Tesseract the primary limitation is the reliance on historic information from the software repositories of the project, which hampers its ability to detect real-time changes to socio-technical congruence and hence its accuracy.

Other awareness tools that relate closely to our approach are those that leverage live workspace information, typically, to address the issue of code dependencies awareness For example, Palantir uses notifications to keep a developer abreast with what happens in her colleagues' workspaces. Notifications relate mainly to artifact changes, and the system delivers notifications to the developers only if they regard some of the same artifacts she has in her workspace [30]. Palantir thus provides each developer with timely but implicit information about any arising same-artifact conflicts, which can be seen as a narrow subset of Coordination Requirements. CollabVS has a somewhat similar focus, but a wider scope, since it encompasses more workspace actions, such as artifact consultation, and also considers a subset of syntactical dependencies between artifacts in its model of interest [13]. CollabVS also has two modes of awareness propagation: a passive one, which presents a developer with a visual panel reporting what her colleagues are doing at the moment; and a proactive one, which uses pop-up notifications, but only when an specific editing conflict is detected. CollabVS in fact deals with CRs as they arise; however, its consideration of workspaces is not intended to produce a complete and explicit view of CRs within a team or throughout a process. It's purpose is to highlight single cases of conflict. FASTDash is another tool for code dependencies awareness for small teams [2]. Similarly to the passive mode of CollabVS, it visualizes what every developer is doing in her workspace in a dashboard that is common to the whole team. FASTDash, though, does not try to establish any work dependencies based on the activities of developers and the technical relationships between artifacts.

Finally, our work has similarities in its goals with other approaches that choose to provide awareness of collaboration issues within development teams by doing away with the usual technical dependencies in favor of different relationships. In one such approach, Storey et al. [33][34] explore the use of shared waypoints and social tagging within code – essentially adding metadata to "TODO" tags to denote the possibility of coordination needs. Such an approach, of course, requires at least one developer who is aware of the coordination need and provides the appropriate tagging so that others may be made aware of it and react accordingly. In contrast, our approach hopes to be able to detect CRs automatically.

# 3. LEVERAGING TASK CONTEXTS FOR COORDINATION

## 3.1 Task Context

The main goal of our work is to identify CRs not only accurately but also early enough to enable decisions on how to best resolve those work dependencies. For example, if Alice and Bob are assigned to tasks that require extensive negotiation and synchronization around interdependent software artifacts, an early design decision to modularize those dependencies can improve Alice's and Bob's individual productivity.

Therefore, we consider ways to capture meaningful data from development work as it happens, so that we can build an incremental record of the development activities carried out on the working set of artifacts for a task. Several facilities of this kind have been described in the literature [30]. Here we focus on Mylyn, a tool that captures individual developer context data as it happens [23][24]. Mylyn is a plugin for the Eclipse IDE whose goal is to provide an individual developer with a task-centric interface. It adapts the Eclipse GUI and focuses its presentation on what is most relevant for that developer in the context of the task she is performing. To achieve that, Mylyn stores data (known as "task contexts") which describe sequences of significant GUI events such as software elements manipulation, artifacts consultation, or issuing of commands. Those events are weighted according to a model of interest that takes into account – among other things – action type, frequency of interaction, and a temporal decay factor. Mylyn task contexts therefore characterize a task in terms of its working set, the relative importance of artifacts in the working set, and the nature of the interactions with those artifacts. Since this kind of data is of general significance, Mylyn has started to extend to a number of other prominent development environments besides Eclipse.

A task context in Mylyn is a list of actions by the developer, described in an XML dialect. The pertinent information associated to each action includes:

- **Kind**: type of interaction (selection, edit, command, etc.)
- **Structure Handle**: a unique ID that identifies a software artifact manipulated by the developer through the Eclipse GUI. Granularity can vary. We can thus choose to consider artifacts at different granularity levels, for example, files vs. classes vs. class elements (methods and attributes).
- **Start Date:** a timestamp
- **End Date** a timestamp, significant only for actions that are not instantaneous

Note that the original purpose of Mylyn task contexts is to support and focus the work of an individual developer throughout the course of a single task or to allow a single developer to easily

switch between tasks. We use the same information to support, instead, a collaborative goal. We want to detect areas in the project that are relevant for the coordination of concurrent tasks by looking at the amount of overlap between the working sets of the developers assigned to those tasks. To achieve that, we introduce a relationship and a measure – which we have named *closeness* – that applies to task contexts and – by extension – to tasks and developers.

## 3.2 Closeness

Our closeness measure conveys the amount of similarity between the development activities recorded in any two task contexts; it is a relative and not an absolute measure. Our algorithm for computing closeness has two conceptual steps and is outlined in Figure 2. First, it computes the union between the artifact sets appearing in the two task contexts. Second, for each element in the union, it considers the kinds of actions recorded in both task contexts for that artifact and applies a weight to that artifact's closeness contribution.

There are several types of actions captured by Mylyn. For this study we only consider **select** and **edit** actions. Other kinds of actions used within Mylyn, such as prediction, propagation and manipulation, were purposely left out of our algorithm. Since these event types are specific to Mylyn, including them would make the replication of our experiments and findings outside of the Mylyn framework difficult. Manipulation actions represent information that developers can explicitly provide to the Mylyn framework to emphasize the importance (or lack thereof) of a given artifact for the task at hand. Prediction and propagation events occur when Mylyn identifies artifacts that have not been included in a developer's working set but appear to be structurally relevant or are returned by Mylyn's Active Search.

We base our weights on the factors that Mylyn itself uses when computing its Degree-Of-Interest (DOI) model for each artifact present in a task context [23]. The DOI provides Mylyn with a way to prioritize the presentation of elements in its task-based interface. The factors used in the Mylyn DOI model are 1 for select events and 0.7 for edit events. The Mylyn factors have been validated empirically by its user community.

Since an edit event in the GUI is always preceded by a selection event, our algorithm combines those events and weighs an edit event as 1.7. An edit overlap occurs when both developers edit the same artifact so each developer contributes 1.7 to the overlap score for a total score of 3.4, which is the maximum score for an overlap. In a mixed overlap, one developer edits the artifact while the other developer selects or views the same artifact. In this case, one developer provides a score of 1.7 for the edit event and the other developer provides a score of 1 for the selection event for a total score of 2.7. In a selection overlap, neither developer edits the artifact but both developers select and view the artifact. In this case, each developer contributes a score of 1 for a total score of 2. We then calculate our weights as percentages of the maximum possible overlap score:

- Edit overlap: 1.7 + 1.7 = 3.4, 3.4/3.4 = 1
- Mixed overlap: 1.7 + 1 = 2.7, 2.7/3.4 = .79
- Selection overlap: 1 + 1 = 2, 2/3.4 = .59

As shown in Figure 2, with those weights we compute two scores: an actual similarity (or overlap) and a potential similarity. The latter indicates the maximum similarity score the two task contexts could reach, if the actions recorded in the two task contexts for the artifacts in their intersection would perfectly match. Our closeness measure is the ratio between the actual and the potential similarity of the two task contexts.

```
CLOSENESS(taskContext1, taskContext2)
1   joint set = union(taskContext1,taskContext2)
2   for each artifact A in joint set
3      if both developers edited A
4         actualOverlap = actualOverlap + 1
5         potential = potential + 1
6         overlapEvents = overlapEvents +
          min(developer X artifact A edit events,
          developer Y artifact A edit events)
7      else if one edited and one selected
8         actualOverlap = actualOverlap + 0.79
9         potential = potential + 0.79
10        overlapEvents = overlapEvents +
          min(developer X artifact A select
          events, developer Y artifact A select
          events)
11     else if both developers selected A
12        actualOverlap = actualOverlap + 0.59
13        potential = potential + 0.59
14        overlapEvents = overlapEvents +
          min(developer X artifact A select
          events, developer Y artifact A select
          events)
15     else if only one developer edited A
16        potential = potential + 1
17     else if only one developer selected A
18        potential = potential + 0.59
19     end if
20  end for each artifact
21  closeness = actualOverlap / potential
```

**Figure 2: Pseudo-code of closeness algorithm.**

Computing closeness only for pairs of contexts is of limited practical use for establishing CRs between developers. However, the closeness relationship and measure can be immediately extended from context pairs to task pairs (there may be multiple contexts per task, for instance because of multiple developers working on the same task) and also developer pairs (by considering all task contexts produced for all tasks to which each developer has contributed in some time frame of interest, such as a software release). When aggregating task context events we apply a scaling factor as a third step to our algorithm. That step considers the total number of actions recorded within all task context records taking part in the aggregated closeness score of each pair (a.k.a., *overlapping events*). We compute the ratio between the number of overlapping events for each pair and the average number of overlapping events computed across all pairs in the population. We use the result as a scaling factor. The rationale for our scaling factor is to place greater weight on those pairs engaged in complex development activities, where coordination is likely to be more necessary than for pairs involved in simple tasks. When performing a complex task, a developer is likely to spend a lot of time on the task and will therefore create many context events. Conversely, a developer completing a task via a trivial change will likely produce a small number of events.

## 3.3 Research Objectives

This study explores the extent to which our closeness measure enables the detection and management of CRs between software developers. Having introduced the semantics of closeness and its measurement technique, we now state our main research objectives:

**Hypothesis H0**: information about task contexts and their closeness relationships – when aggregated at the developer level – will provide a view of CRs that is analogous to that offered by using information about artifact commits and technical dependency relationships. H0 is our principal hypothesis and speaks to the feasibility of detecting CRs by leveraging developers' working sets and their overlap (and bypassing the analysis of technical dependencies between artifacts).

**Hypothesis H1**: basing CR detection on task contexts and closeness can provide more accurate results than current methods. H1 speaks to the ability of leveraging the detailed information offered by task contexts to improve either precision, recall, or both in detecting CRs.

**Hypothesis H2**: the "live" data provided by task contexts supports the detection of CRs substantially earlier than current methods. H2 speaks to the possibility of supporting timely decisions on how to handle CRs as they form in the project.

# 4. CASE STUDY – MYLYN

The hypotheses stated in Section 3.3 summarize our research objectives. In order to verify these hypotheses, we have carried out an empirical study on the open source project handling the development of the Mylyn framework itself. Contributors to the Mylyn Eclipse plugin use Mylyn for their work and routinely publish the task context data in the project's Bugzilla repository. There are 69 other projects in the Eclipse community reporting freely available Mylyn task contexts, but the Mylyn project provided a case with the most widespread and complete task context information.

## 4.1 Data Collection and Preparation

We collected development data over eight releases of Mylyn (v2.0 through v3.3). These releases represent almost three years of development, from December of 2006 until October of 2009.

We focused our data collection on the 1,970 tasks which have associated task context data. For those tasks, we collected not only the context files, but the whole content of the Bugzilla entry including *patch description files* which are attached by those developers who do not have commit privileges. Those patch descriptions report the *diff* information for all artifacts which were modified as part of a patch and are essentially equivalent to commit records kept in the code repository of the project (SVN). We also collected the logs for all changes submitted by developers with commit privileges to the SVN repository of Mylyn for the period of interest. In this data set, we found 51 distinct developers who attached task context data (*context attachers*) and 8 distinct developers with commit privileges (*committers*). Each individual release has between 10 and 32 distinct context attachers and 4 to 6 distinct committers.

With respect to task context data, we collected 588,796 events related to select and edit actions within the 1,970 tasks considered. Many of these events pertain to .class, .jar, and other file types which are by-products – rather than subjects – of development work. For the rest, the large majority of events relate to source code (Java) files. We focused on the 450,747 context events dealing with Java artifacts over the eight releases of interest.

We filtered the commits to include only java files, which yielded 27,074 single file commits over the same time period. Most of the commits, 25,135 or 92.8%, were linked directly to 1,835 unique tasks via the Bugzilla task IDs conventionally inserted in the commit comments. We then intersected those tasks with the set of tasks with associated context data, which yielded for our case study a data set of 1,127 tasks, 10,647 single file commits and the 450,757 context events mentioned above. We called this first data set DS1.

Upon further examination of DS1, we noticed that a number of file changes reported in commits for a given task were not matched by any of the edit events in that task's context data. In the time period considered, we identified 6,507 single file commits (out of 10,647) that are not matched by any proof of editing of the same files in the associated task contexts by the developer who committed the change. The reason is that, although it is customary and usual to submit a context file when committing a change for a task, developers in the Mylyn project do not always abide to this convention. Another possible reason for this is the developer who committed the change is not the developer who contributed the patch. Therefore, we split the DS1 data set in two: DS1-a includes only the 4,140 commits for which we have matching events within task contexts; whereas DS1-b includes the other 6,507 commits.

Since looking at just commit data, as in DS1, would limit our analysis to the CRs only between the few developers with commit privileges, we also looked at patch description files attached to Bugzilla records for the Mylyn releases taken into consideration. The files contain data on 7,196 java file changes that span 936 unique tasks. The task contexts for those yielded 345,521 context events for java artifacts, contributed by 74 distinct developers.

Only a portion of the patch description files retrieved from Bugzilla are matched to a task context, yielding 1,387 file changes with associated task context data from 34 contributors. We proceeded to check whether this set of changes is disjoint from those reported by commits and included in DS1. There is only one committer in this set, who is responsible for 219 changes. A manual inspection revealed that only 11 of those 219 changes overlap with commits made by the same developer in DS1. The 1,387 patch file changes above therefore represent different development work from what is captured in DS1. It can be noted that not all of the 1,387 file edits in this set were ultimately committed to the code base, but for our purposes that is unimportant since we are looking for coordination requirements which existed at the time of actual development work. Any files which have been edited as part of a patch provided by a contributor represent actual development work regardless of whether or not that patch was accepted or not. These 1,387 patch file edits along with the 345,521 associated task context events, have become our second data set DS2.

Finally, we combined DS1-a and DS2 into a third data set DS3, including all records of file changes (either via commit traces or patch diff files) for which have corresponding task context data.

**Table 1. Data Sets**

| Data Set | Description |
|---|---|
| DS1-a | 4,140 commits with 450,757 associated task context events. |
| DS1-b | 6,507 commits with 450,757 unassociated task context events. |
| DS2 | 1,387 patch file edits along with the 345,521 associated context events. |
| DS3 | Commits and patch file edits with associated task context data (DS1-a and DS2 combined). |

## 4.2 Method of Analysis

We have leveraged the data sets described in Section 4.1 to investigate our research hypotheses. We looked into H0 and H2 quantitatively and H1 both quantitatively and qualitatively.

For H0, we used the data at our disposal to compute CRs using the method proposed by Cataldo et al. We then correlated statistically the results of that method with closeness scores for the corresponding developer pairs, to examine whether and to what extent closeness between developers can be considered a proxy for the presence of CRs.

For hypothesis H1, we examined issues of precision and recall between the traditional CR detection method and our closeness measure. We sampled cases of mismatches between the two methods. We then analyzed them in detail to understand the possible causes of those mismatches.

For hypothesis H2, we gauged closeness as an antecedent of CRs. We looked at pairs of developers with CRs to identify the earliest moment in which their context overlaps contribute to closeness and provide evidence of a work dependency within the same Mylyn release.

## 4.3 Results

### *4.3.1 H0: Is Closeness a Good Proxy for Coordination Requirements?*

We started our analysis with data set DS1-a. We used the 4,140 commits in that set to calculate CRs between committers according to the method described by Cataldo et al. [8]. To define technical dependencies between the involved software artifacts, we used the "files committed together" heuristic introduced by Gall [17] to define logical coupling between artifacts as recommended in [6]. We computed our closeness scores for the same set of committers, using a single file as the unit of granularity. When computing both CRs and closeness, we looked at concurrent work in each release separately which resulted in 70 committer pairs.

Firstly, we checked that higher values of closeness correlate with the likelihood of a CR, by performing a point-biserial correlation with a binary vector denoting the presence of CRs. We then performed a Spearman correlation between the count of CRs for each pair and their closeness scores. We chose to use a Spearman correlation because both the CR counts and closeness scores are not normally distributed. Both tests were statistically significant and provided us with strong positive correlations, as shown in Table 2. Moreover, we observed that 46 of the 70 committer pairs had some CR, and 43 of those 46 pairs (93.5%) have closeness > 0. Conversely, of the 24 pairs with no CRs, 7 present a closeness score of 0.

**Table 2. DS1-a CR v Closeness Correlations**

| Test | p-value | rho |
|---|---|---|
| Spearman | 2.405e-11 | 0.6877365 |
| Point-biserial | 4.865e-07 | 0.5467687 |

We repeated the same test also by computing closeness not at the file level, but at the finest granularity level for artifacts reported in Mylyn context events. Since traditional CRs are determined at the file level, our expectation was that analysis of contexts at finer granularity would yield less closeness and lower levels of correlation. The results shown in Table 3 are in line with those expectations, but correlations between closeness and CRs are still strong and significant. Of the 46 CR pairs, we found that 42 (91.3%) have closeness > 0. Therefore, the additional granularity removes closeness from 1 of the 43 pairs with closeness when looking at single file granularity. 8 pairs have a granular closeness score of 0.

**Table 3. DS1-a CR v Granular Closeness Correlations**

| Test | p-value | rho |
|---|---|---|
| Spearman | 6.803e-09 | 0.6161813 |
| Point-biserial | 8.76e-06 | 0.4889009 |

These findings seem to confirm that closeness is a valid proxy for CRs and our main hypothesis, H0. However, since by construction DS1-a contains rather homogenous commit and context data (artifacts that are committed are likely to show up prominently in select and edit actions within task contexts), we decided to reinforce those findings by performing analogous tests on DS1-b. It is worth underlining that in this setting the CR and closeness analyses are performed on two completely distinct set of artifacts. Since we are still looking at manifestations of the work done by the same developers on the same set of tasks, we speculated that closeness scores should remain a good indicator of CRs.

The results of our tests on DS1-b, shown in Table 4, show strong and statistically significant positive correlations. Among the 75 developer pairs showing up in this data set, 33 have a CR and all 33 pairs have a closeness score > 0 at both the file and granular level of analysis. Of the 42 pairs with no CRs, 10 of the pairs also have a closeness score of 0 at the file level and 14 at the granular level.

**Table 4. DS1-b CR v Closeness Correlations**

| Test | Unit of Work | p-value | rho |
|---|---|---|---|
| Spearman | File | 8.738e-09 | 0.595492 |
| Point-biserial | File | 1.109e-08 | 0.5920081 |
| Spearman | Granular | 2.500e-07 | 0.5423827 |
| Point-biserial | Granular | 1.770e-07 | 0.5482931 |

Finally, we further validated H0 by moving our analysis from the restricted core group of Mylyn committers of data set DS1 to the larger group of project contributors who submitted a patch and an associated task context, represented in data set DS2. That data set includes 277 developer pairs, of which 37 have CRs computed via the traditional pairs. Of these, 28 (75.7%) have file-level closeness > 0 and 24 (64.9%) have granular closeness > 0. Of the 240 pairs with no CRs, 206 also have a closeness score of 0 at the file level and 222 at the granular level. Table 5 shows strong positive correlations that are also statistically significant.

**Table 5. DS2 CR v Closeness Correlations**

| Test | Unit of Work | p-value | rho |
|---|---|---|---|
| Spearman | File | < 2.2e-16 | 0.5519499 |
| Point-biserial | File | < 2.2e-16 | 0.5361256 |
| Spearman | Granular | < 2.2e-16 | 0.5656959 |
| Point-biserial | Granular | < 2.2e-16 | 0.5510784 |

We also report the results for the combined DS1-a and DS2 data sets. As previously discussed, these data sets represent disjoint development activities and have no overlapping developer pairs. Therefore, their union provides us with our largest data set DS3, which includes a total of 347 developer pairs. Table 6 shows the CR to closeness correlations for DS3.

**Table 6. DS3 CR v Closeness Correlations**

| Test | Unit of Work | p-value | rho |
|---|---|---|---|
| Spearman | File | < 2.2e-16 | 0.683612 |
| Point-biserial | File | < 2.2e-16 | 0.6558573 |
| Spearman | Granular | < 2.2e-16 | 0.6815387 |
| Point-biserial | Granular | < 2.2e-16 | 0.6551155 |

### *4.3.2 H1: How Accurate is Closeness in Determining Actual Coordination Requirements?*

To investigate H1, we first reviewed how developer pairs with CRs match against pairs with closeness >0, and conversely how developer pairs with no CRs match against pairs with closeness = 0. We have outlined those results in the discussion in Section 4.3.1, but we examine them here in terms of CR precision and recall.

Results at the granular level are reported in Table 7 and assume that CRs detected with the traditional method are the ground truth. In that framework, although precision and recall vary across the various data sets, results are mostly satisfactory. Some results, in particular for DS1-a and DS1-b, also suggest that matching CRs to closeness at the 0-level may cast a net that is too wide. As part of our future work, we will carry out a sensitivity analysis to gain insight on this matter and understand the impact and appropriateness of different closeness thresholds for CR detection. However, for the purpose of our comparison to the traditional method a threshold of 0 seems satisfactory since the lowest possible CR score of 1 indicates that a developer pair worked on only one pair of dependent files. The lowest closeness score of 0.01 also indicates at least one dependency.

**Table 7. Precision and Recall (Granular)**

| Data Set | # of pairs | Precision | Recall |
|---|---|---|---|
| DS1-a | 70 | 42/58 = 0.724 | 42/46 = 0.913 |
| DS1-b | 75 | 33/61 = 0.541 | 33/33 = 1 |
| DS2 | 277 | 24/40 = 0.6 | 24/37 = 0.649 |
| DS3 | 347 | 70/100 = 0.7 | 70/97 = 0.722 |

Since CRs computed with the traditional method are themselves only an approximation of ground truth, we then proceeded to manually examine some mismatch cases. Our goal here was to determine if the richer representation of artifacts and work leveraged in our algorithm is conducive to identify actual CRs more accurately by weeding out either false positives or false negatives of the traditional method.

For potential false positives we reviewed the four cases in DS1-a of developer pairs with CRs and granular closeness = 0. The record of changes made by each pair of developers held in the relevant task contexts determined that in each case not only did the developers consult and make changes to a totally disjoint set of files, but also all of the recorded code changes were in areas of those files that appear unrelated to one another. In DS1-a, a simple example is provided by the single CR that exists between developers 6 and 7 in release 3.2. Granular level closeness for this pair is 0. Developer 6 committed `BugzillaClient.java`, while developer 7 committed `BugzillaTaskEditorPage.java`. The changes by developer 6 involve a character encoding method that is private to the `BugzillaClient` class, whereas developer 7 added a new section to the Mylyn task editor. Although those changes are effectively unrelated, the two involved files were changed together by other developers during the same release often enough to cause a logical dependency to be established by the CR detection algorithm. We noticed analogous incidents in the other three cases represented in DS1-a. Those CRs are therefore false positives of the traditional method that are correctly not picked up by our closeness algorithm.

Moving to potential false negatives, we examined the 16 committer pairs in DS1-a which present some amount of granular closeness but have no CR. Our review allowed us to recognize two distinct, interesting types of behavior in this set.

In a single case, involving developers 3 and 7 during release 3.3, closeness contributions came exclusively by selection and mixed overlaps. The pair in fact had seven mixed overlaps and six selection overlaps. Meaning that developers 3 and 7 worked on 13 of the same artifacts, of which seven were edited at some point by either developer 3 or developer 7 but no single artifact was edited by both developer 3 and developer 7. No logical dependencies exist between these seven artifacts which were edited by either of the developers, and, since there were no overlapping commits, the traditional CR method does not allow for a CR to be detected. However, our closeness algorithm does pick up on a small dependency here since we have the advantage of knowing not only what files are edited by a developer but also what files are viewed by a developer in the process of completing a task. Our algorithm picks up what is likely an actual work dependency, since developer 3 and developer 7 repeatedly viewed the same area of the software code base and viewed the work done by each other during their work for release 3.2. The dependencies picked up because of this information are likely not to be large since no overlapping files are being edited, but it does provide additional information which is missed by the traditional algorithm.

The remaining 15 developer pairs we examined represent an even more interesting case. In each of these cases, the developer edit from 1 to 67 of the same artifacts (17 on average). CRs could not be established in any of these cases because at least one of the two developers did not commit her changes in the end. However, task contexts prove that those developer pairs were at one time engaged in concurrent development on the very same artifacts, which is the epitome of a Coordination Requirement. We imagine that in at least some of those cases, the two committers became somehow aware of the overlap, and they avoided conflict by having one of the two merge all changes and commit on behalf of both. Evidence of such a scenario may be recorded in the archived communications for the Mylyn project which we intend to mine in the future.

All of the cases we examined in DS1-a, therefore, turned out to be false positives or negatives of the traditional CR detection method. More importantly, they demonstrate the drawbacks of the method's reliance on post-mortem information and technical dependency representations and heuristics that may be irrelevant or incorrect. We foresee that a number of analogous cases can be uncovered also in other data sets, proving further that our closeness relationship is an accurate indicator of CRs.

### *4.3.3 H2: Does Closeness Provide Timely Detection of Coordination Requirements?*

Finally, we set out to investigate whether closeness – besides being accurate – is also an early indicator of CRs. Since context events used by our algorithm, by their nature, are antecedents to the commits used in the traditional method, our analysis focuses on how early our method provides evidence of work dependencies between concurrent tasks. The earlier the evidence, the more

actionable it is in supporting decisions aimed at resolving CRs as they form.

For this analysis, we used the two data sets for which we have task context data associated with file changes (DS1-a and DS-2). In those data sets, we considered all pairs of developers who present some CR while working on some release and have granular closeness >0. In DS1-a, there are 42 such pairs while in DS2 there are 24.

We then looked at the day when the first contribution to the closeness score occurs. We compute this by examining the timestamps for the first event recorded in all tasks contexts for each of the two developers for that release and picking the latest of the two timestamps. We then compare the first closeness event with the first day of concurrent work by that pair during the release of interest.

We found that in DS1-a the first evidence of closeness is detected on average 14.2 days after parallel work begins. In DS2, it takes 5.8 days. To put these findings in perspective, we considered the days in which the first CR is identified for the same pairs. That happens in DS1-a 60.7 days on average after the beginning of concurrent work by a pair (that is, with a delay of 46.5 days). In DS2, the first CR is detected 16.9 days after the concurrent work begins (that is, 11.1 days later). We also compared our findings with the duration of the concurrent work intervals by the same pairs in the various releases. In data set DS1-a, concurrent work intervals last 102 days on average, whereas in DS2, they last 31.2 days on average. Therefore, the average "advance notice" provided by closeness amounts to 87.8 days and 25.4 days, respectively.

Since these results were encouraging in terms of early recognition of work dependencies that lead to CRs, we also looked at how early closeness manifested across all developer pairs who performed concurrent work on either the same task or tasks overlapping in time within a release. We considered all task contexts for the eight releases, which yielded 3,013 such developer pairs. For those pairs, the first evidence of closeness occurred on average 1.7 days after the start of concurrent work. Moreover, in the majority of cases, 2,713 out of 3,013, the first evidence of closeness occurred on the same day as the beginning of concurrent work. That can be contrasted with the average task duration of 15.5 days. Here, the task duration is determined by the first context event to the last context event for any given task so as to only count the actual development time of a task.

While our analysis was of course done *post hoc*, these findings indicate that a tool that uses task context events in real time could produce closeness scores which will provide timely detection of Coordination Requirements.

# 5. DISCUSSION

## 5.1 Applications

The results discussed in Section 4.3 have numerous implications. First, they suggest that Coordination Requirements can be determined accurately based exclusively on the similarity (or closeness) of task contexts. Our algorithm works independent of any conceptualization of technical dependencies between tasks or software artifacts. Therefore, unlike traditional methods which rely on information that becomes available only after work has been completed, our algorithm relies on data that is accessible while development work is under way. This allows for earlier detection of CRs. Task context information is not only timelier, but also richer and more precise. It captures the entire set of artifacts involved in a task, including artifacts that are finer-grained than a file. It follows the evolution of that working set during the task duration and it tracks how artifacts are actually used.

A further implication is the ability to rank (and hence prioritize) detected CRs based on the corresponding closeness measures. The traditional method of CR detection provides a form of ranking, simply through the count of CRs occurring within a pair of developers. Our rank is based instead on two components: the amount of weighted overlap in the working sets of the pair and the number of overlapping context events as compared to the average number across the population. The ranking based on the amount of overlap in working sets is analogous to the ranking based on count of CRs. However, ranking based on the number of overlapping events allows our algorithm to de-emphasize trivial development activities and their impact on the need to coordinate.

This kind of information about CRs, which is at once timely, rich, dynamic, and granular, can have important consequences on the construction of tools. A method that not only computes CRs in real time as they form, but also enables tracking their evolution over time and the continuous ranking of their importance, can be used to support enhanced awareness in a CSCW environment. It can also be incorporated in a decision support tool. For example, it can provide a dynamic view of Socio-Technical Congruence in a project to inform real-time decisions on aspects like tasks assignment, scheduling, team composition and layout, or design refactoring, as proposed by Valetto et al. [37].

While the analysis presented in this paper focuses on capturing the Coordination Requirements between pairs of individuals, closeness can easily be applied at other aggregation levels such as tasks, projects or teams simply by rolling task context information up to the desired level. It is also possible to move away from dyadic analysis altogether. For example, our method can support identification of emergent teams within a software development organization, based on the context overlap they have and the corresponding ties measured through the closeness relationship. Such analysis of emergent teams would enjoy the same characteristics (real-time applicability, dynamism and prioritization) discussed above.

## 5.2 Threats to Validity

A general caveat about our findings is, of course, that they derive from a single project, which limits their reliability. Those findings should be corroborated by additional empirical studies. Since in this case study the data sets are moderate in size, in particular with respect to the number of developers involved and the number of CRs, it would also be important to repeat our study on larger software organizations with denser work dependencies networks.

A limitation can derive from our choice to perform our analysis at the release level. When considering concurrent work at finer-grained temporal units, the outlook on CRs and/or closeness may differ. To properly investigate how sensitive our findings are with respect to this issue, a project with a rather high density of CRs would again make for the best case study.

Another limitation of our study is that we have somewhat arbitrarily considered any level of closeness score >0 as an indicator of possible CRs. We think that, for the purpose of our comparison to the traditional method, that is the correct choice. The lowest possible CR score of 1 indicates that a developer pair worked on only one pair of dependent files. Analogously, the lowest closeness score (0.01) between two developers indicates at least one overlap in their working sets. However, a sensitivity

analysis on this case study, or – better – across multiple case studies, is necessary to confirm (or disprove) our reasoning.

Finally, there may be issues of repeatability. Firstly, although the adoption of Mylyn is very wide and ever-increasing in open source as well as industrial settings, its consistent use by all developers in a project during all (or at least the majority) of the project activities is not guaranteed. Moreover, although researchers and practitioners could likely leverage other similar commercial-grade facilities (such as Cubeon for the Netbeans IDE[1]), they cannot assume that they report the same exact kind of information made available by Mylyn. Therefore, when dealing with other similar technologies, it may be necessary to adapt the closeness algorithm accordingly, which in turn may require its re-validation along the same lines described in this paper.

## 5.3 Future Work

In the short term, we intend to further leverage the Mylyn data set to gain further insight on the potential of contexts and closeness. We also plan to pursue the resolution of the limitations discussed in Section 5.2, with this and other retrospective case studies.

The next natural step is to develop and evaluate a tool that leverages task contexts and the closeness algorithm to detect coordination requirements as soon as they form to assist the coordination of development teams. Such a tool could make developers and project leaders aware of impending CRs and rank them in real time. Given the rich information at the basis of our closeness measure, it could also highlight the artifacts involved in each CR and make developers' actions on those artifacts translucent [16], introducing a potential for implicit coordination. Such a tool would provide a basis for managers and developers to make decisions on how to best resolve CRs, especially when early CR detection is coupled with real-time computation of socio-technical congruence.

We plan to evaluate this tool with a controlled experiment, which will also provide us with a "live" case study to carry out further empirical validation of our method. For example, by taking daily snapshots of the closeness scores generated by the tool, we will be able to assess precisely the level of early CR detection supported by our tool. That rich data set will also allow us to determine experimentally the appropriate closeness threshold for making the tool users aware of impending CRs.

# 6. CONCLUSION

We have introduced a closeness relationship that can be used to infer Coordination Requirements between software developers as they form. We have also described an algorithm for measuring that closeness, based on task context information which details activities of developers within their individual development environments and can be easily obtained with existing tools. We have shown that closeness thus measured is an accurate and early indicator of Coordination Requirements and can overcome the known drawbacks and imprecision of current methods for the detection of Coordination Requirements. The techniques we propose are promising for building support to management-, design- and team-related decisions that need to be made to properly manage coordination within a software project and organization while the development work is under way.

[1] See http://code.google.com/p/cubeon/

# 7. ACKNOWLEDGMENTS

We would like to thank Dave Berry for his help with the initial data mining for this paper. Special thanks to Gail Murphy for initial discussions and her constant advice and encouragement on this topic. This work was partially supported by the National Science Foundation through grant no. CCF-0916891.

# 8. REFERENCES

[1] Baldwin, C. Y. and Clark, K. B. 2000 Design rules: The power of modularity. The MIT Press.

[2] Biehl, J., et al. 2007. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems.* CHI'07,1313-1322.

[3] Brooks, F. P. 1995. The Mythical Man-Month: Essays on Software Engineering. Addison Wesley. Reading, MA.

[4] Browning, T. R., Co, L. M. A., and Worth, F. 2001. Applying the design structure matrix to system decomposition andintegration problems: a review and new directions. *IEEE Transactions on Engineering management*. 48, 3, 292-306.

[5] Cataldo, M., Bass, M., Herbsleb, J., and Bass, L. 2007. On Coordination Mechanisms in Global Software Development. In P*roceedings of the International Conference on Global Software Engineering*. ICGSE 2007, 71-80.

[6] Cataldo, M., Herbsleb, J. D., and Carley, K. M. 2008. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ESEM 2008, 2-11.

[7] Cataldo, M., Mockus, A., Roberts, J. A., and Herbsleb, J. D. 2009. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*. 35, 6, 864-878.

[8] Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, CSCW 2006.

[9] Conway, M. E. 1968. How do committees invent. *Datamation. 14, 4, 28-31*.

[10] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S. 2005. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*. 31, 6, 446-465.

[11] de Souza, C. R., Quirk, S., Trainer, E., and Redmiles, D. F. 2007. Supporting collaborative software development through the visualization of socio-technical dependencies. In *Proceedings of the 2007 international ACM conference on Supporting group work*. 147-156.

[12] de Souza, C. R. B., Redmiles, D., Cheng, L. T., Millen, D., and Patterson, J. 2004. How a good software practice thwarts collaboration: the multiple roles of APIs in software development. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of software engineering.* FSE 2004.

[13] Dewan, P. and R. Hegde. 2007. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In *Proceedings of the European Conference*

*on Computer Supported Cooperative Work.* E-CSCW 2007. p. 159-178.

[14] P. Dourish and V. Bellotti. Awareness and Coordination in Shared Workspaces. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, CSCW 1992: p. 107-114.

[15] Ehrlich, K., Helander, M., Valetto, G., Davies, S., and Williams, C. 2008. An analysis of congruence gaps and their effect on distributed software development. In *Proceedings of the ICSE Workshop on Socio-Technical Congruence* STC 2008.

[16] Erickson, T. and Kellogg, W. A. 2000. Social Translucence: An Approach to Designing Systems that Support Social Processes. *ACM Transactions on Computer-Human Interaction.* 7, 59-83.

[17] Gall, H., Hajek, K., and Jazayeri, M. 1998. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*. ICSM 1998.

[18] Gokpinar, B., Hopp, W. J., and Iravani, S. M. R. 2010. The Impact of Misalignment of Organizational Structure and Product Architecture on Quality in Complex Product Development. *Management Science*. 56, 3, 468-484.

[19] Grinter, R. E., Herbsleb, J. D., and Perry, D. E. 1999. The geography of coordination: dealing with distance in R&D work. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work*.

[20] Herbsleb, J. D. and Grinter, R. E. 1999. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st International Conference on Software Engineering*. ICSE 1999, 85-95.

[21] Herbsleb, J. D. and Mockus, A. 2003. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*. 29, 6, 481.

[22] Herbsleb, J. D., Mockus, A., and Roberts, J. A. 2006. Collaboration in software engineering projects: A theory of coordination. In *Proceedings of the 27th International Conference on Information Systems*. ICIS 2006.

[23] Kersten, M. and Murphy, G. C. 2005. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development.* AOSE 2005, 159-168.

[24] Kersten, M. and Murphy, G. C. 2006. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering.* FSE 2006.

[25] Kraut, R. and Streeter, L. 1995. Coordination in software development. *Communications of the ACM.* 38, 3, 69-81.

[26] McDonald, D. W. 2003. Recommending collaboration with social networks: a comparative evaluation. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems.* CHI'03. 600-608.

[27] Minto, S. and Murphy, G. C. 2007. Recommending emergent teams. In *Proceedings of the 4th Workshop on Mining Software Repositories,* MSR 2007.

[28] A. Mockus and J. D. Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In P*roceedings of the 24th International Conference on Software Engineering*. ICSE 2002, 503–512.

[29] Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM.* 15, 12, 1058.

[30] Sarma, A., Noroozi, Z., and van der Hoek, A. Palantír: raising awareness among configuration management workspaces. In *Proceedings of the 25th international Conference on Software Engineering* ICSE 2003.

[31] Sarma, A., Maccherone, L., Wagstrom, P., and Herbsleb, J. 2009. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of the 31st International Conference on Software Engineering*. ICSE 2009, 23-33.

[32] Sosa, M. E., Eppinger, S. D., and Rowles, C. M. 2004. The misalignment of product architecture and organizational structure in complex product development. *Management Science*. 50, 12, 1674-1689.

[33] Storey, M. A., Cheng, L. T., Bull, I., and Rigby, P. 2006. Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of the 20th anniversary conference on Computer Supported Cooperative Work.* CSCW 2006.

[34] Storey, M. A., Cheng, L. T., Singer, J., Muller, M., Myers, D., and Ryall, J. 2007. How programmers can turn comments into waypoints for code navigation. In *Proceedings of the IEEE International Conference on Software Maintenance*. ICSM 2007. 265-274.

[35] Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. 2001. The structure and value of modularity in software design. In *Proceedings of the 9th ACM SIGSOFT international symposium on Foundations of Software Engineering*. FSE 2001, 99-108.

[36] Terveen, L. and McDonald, D. W. 2005. Social Matching: A Framework and Research Agenda. *ACM Transactions on Computer-Human Interaction*. 12, 3, 401-434.

[37] Valetto, G., Chulani, S., and Williams, C. 2008. Balancing the value and risk of socio-technical congruence. In *Proceedings of the ICSE Workshop on Socio-Technical Congruence*. STC 2008.